Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: advanced object-oriented features

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Namespaces

Namespaces

Classes can be grouped in namespaces

- A hierarchical grouping of classes and other entities

- Every source file defines a global namespace
  - possibly implicitly, if the user doesn't provide a name

- May affect visibility (but in general namespace ≠ assembly)

- Unlike Java, there need not be any connection between namespaces and directory structure

- The following are allowed in C# and disallowed in (the official implementation of) Java:
  - multiple public top-level classes in the same file
  - splitting the declaration of a class across multiple files

Using namespaces

- Namespace declaration:

```
namespace MyNameSpace { ... }
```

- Load all classes in a namespace (but not sub-namespaces) with the `using` keyword:

```
using System;
Console.WriteLine("Hi!");
```

  instead of:

```
System.Console.WriteLine("Hi!");
```

- Upon importing you can declare an alias:

```
using MyConsole = System.Console;
MyConsole.WriteLine("Hi!");
```

Partial classes

The keyword **`partial`** denotes classes, structs, and interfaces whose definition is split in multiple parts (possibly in different files) within the same namespace and assembly.

- Modifiers of the parts cannot conflict (e.g., **`public`** vs. **`private`**); if optional, the most general one is assumed (e.g., **`abstract`** vs. non-**`abstract`**).

Possible usages:

- Automatic incremental code generation
- Separation of programmers' work

```
public partial class Employee {  // in file1.cs
     int Salary(int year);  }

public partial class Employee {  // in file2.cs
     string Role();  }
```

BCL: Base Class Library

- **`System`**
  (basic language functionality, fundamental types)
- **`System.Collections`** (collections of data structures)
- **`System.IO`** (streams and files)
- **`System.Net`** (networking and sockets)
- **`System.Reflection`** (reflection)
- **`System.Security`**
  (cryptography and management of permissions)
- **`System.Threading`** (multithreading)
- **`System.Windows.Forms`**
  (GUI components, nonstandard, specific to the Windows platform)

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Abstract classes and interfaces

Abstract classes and interfaces

A class member may or may not have an implementation

- if it lacks an implementation, it is abstract

A class whose implementation is not complete is also called abstract

- but even a fully implemented class can be declared abstract

Interfaces are a form of fully abstract classes

- they enable a restricted form of multiple inheritance

Abstract classes

- An **`abstract`** class cannot be directly instantiated
- An **`abstract`** method cannot be directly executed
- If a class has an **`abstract`** method, the class itself must be **`abstract`**
- An **`abstract`** class cannot be **`sealed`**
- Useful for conceptualization and partial implementations

Interfaces

- Declared using **`interface`** instead of **`class`**
- Equivalent to a fully **`abstract`** class
  - you don't use the keyword **`abstract`** in an **`interface`**
- A way to have some of the benefits of multiple inheritance, with little hassle (e.g., selecting implementations)
- A class may inherit from one or more interfaces
  - If the class inherits from another class **and** some interfaces, the class must come first in the inheritance list
- An interface can also inherit from one or more interfaces

Interface use

- For typing, implementing an interface is essentially equivalent to extending a class: polymorphism applies

- All interface members are implicitly **`abstract`** and **`public`** (and non-static)
- But the interface itself may have restricted visibility

- Interfaces can have: methods, properties, events, indexers
- Interfaces cannot have fields

  This is C#'s way to push programmers to have only private or protected fields

  - What's the principle behind this?

Method name clash in multiple interfaces

Two interfaces **`I1`** and **`I2`** may define two methods with the same name and signature.

If a class **`C`** a extends both **`I1`** and **`I2`**, it can provide two implementations of the method, one for each interface. This is called explicit interface implementation.

- If only one implicit implementation is provided, the two methods are merged (same behavior as in Java)
- Explicit interface implementations are also applicable when there is no name clash
- Methods realizing an explicit interface implementations can only be called using references of the interface's type
- Having both implicit and explicit interface implementations of the same method is also allowed

Explicit interface implementation

```
interface I1 {  int Weight(); }  // in kg
interface I2 {  int Weight(); }  // in lbs

class C : I1, I2 {
  int I1.Weight() { return this.Volume * 3; }
      // roughly, 1 kg = 2 lbs
  int I2.Weight() { return this.Volume * 3 * 2;  }
}

C c = new C();
int w = c.Weight();  // compiler error!
I1 i1 = (I1) c;
I2 i2 = (I2) c;
int kg = i1.Weight();  // in kg
int lbs = i2.Weight();  // in lbs
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Delegates and events

Events and Delegates

C# provides language features for event-driven programming

- most common application: GUI programming

Delegates are object wrappers for operations

- similar to C/C++ function pointers, but with type safety
- similar to Eiffel's agents
- similar functionality achieved in Java with anonymous inner classes

Events are signals sent by an object to communicate the occurrence of an action

- event publisher: object which can signal an event
- event subscriber: object which triggers some action when an event is signalled
- multicast communication applies
  - an event can have multiple subscribers
  - a subscriber can subscribe to multiple events

Delegates

Delegates are object wrappers for operations

- They can be declared anywhere as members in a namespace, including outside any class
- The declaration includes a return type, a name, a list of arguments

```
public delegate void BinaryOp (int i, int j);
```

- This is a placeholder for methods taking two integer arguments and returning none

Delegates

After being declared, delegates can be instantiated by passing a handler to an actual method implementation

- The signature (and return type) of the passed method must match that of the delegate

```
BinaryOp bop = new BinaryOp(adder.AddPrint);
```

adder.AddPrint references method **AddPrint** of object attached to reference **adder**.

- You can attach (and remove) multiple methods to the same delegate, or attach the same method multiple times

```
bop += new BinaryOp(multiplier.MultPrint);

BinaryOp b2 = StaticMethodOfCurrent +
           new BinaryOp(adder.AddPrint) +
           new BinaryOp(adder.AddPrint);
b2 -= StaticMethodOfCurrent;
```

Delegates

After instantiation, a delegate can be invoked

- the net effect is equivalent to calling synchronously the passed method(s)

```
bop(3, 5);      // prints 3+5 and 3*5
```

- if multiple methods are attached to the delegate, their order of execution is nondeterministic
- if the attached methods return a value, the call through the delegate returns the last computed value

Other methods are available to control the invocation order and use multiple returned values

```
foreach (BinaryOP b in bop.GetInvocationList()) {
    b(3, 5);    // single invocation
}
```

Events

Events are signals sent by an object to communicate the occurrence of an action

- An event is a member of some class
- The declaration associates an event name to a delegate type

```
public event BinaryOp BOPRequest;
```

- Any class that can trigger the event will have a "trigger method" for the event
  - naming convention for the trigger method: `OnEventName`

```
void OnBOPRequest(int i, int j) {
  if (BOPRequest != null) { BOPRequest(i,j); }
}
```

Events

Subscribers to an event provide a handler for that event in the form of a method

- They register it on the event using the delegate type associated to the event

```
BOPRequest += new BinaryOp(adder.AddPrint);
BOPRequest += new BinaryOp(multiplier.MultPrint);
```

- Whenever the event is triggered, all the registered methods of the subscribers are executed synchronously

```
OnBOPRequest(3, 5); // prints 3+5 and 3*5
```

- Delegates provide a mechanism to decouple event generation and handling: the writer of the event class doesn't know what handlers will be attached to it

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

“Special” classes and features

The String class

Sequences of Unicode characters

- `string` (all lowercase) is an alias for `String`
- Immutable class: no setters

Some differences w.r.t. Java:

- `==` and `!=` pre-defined to compare string content, not addresses
- Individual characters accessible with array notation

```
Console.WriteLine(“Hi!”[2]);  // prints: !
```

Two formats for constant strings:

- quoted: escape characters are processed

```
String s = “A \”path\” c:\\myDir\\onWindows”;
```

- @-quoted: escape characters are not processed

```
String s = @“A ““path”” c:\myDir\onWindows”;
```

Object comparison: `Equals`

```
public boolean Equals(Object obj) {
    return (this == obj);
}
```

- The default semantics compares addresses

- We can provide a different semantics by redefining it
  - What kind of redefinition is appropriate (overriding or shadowing)?
  - Implementation should be an equivalence relation
    - Reflexive, symmetric, transitive
  - For any non-**`null`** reference variable **`x`** it should be:

    ```
    x.Equals(null) == false
    ```

- It is usually necessary to override **`GetHashCode()`** as well, because equal objects should have equal hash codes

Class `Object`: hash code

```
public virtual int GetHashCode()
```

The default implementation of `GetHashCode()` does not guarantee that different objects return different hash codes.

In general, it is necessary to override `GetHashCode()`, so that equal objects have equal hash codes.

Overriding `Equals()` in descendants does not guarantee to give the right semantics to `GetHashCode()` as well.

Class `Object`: string representation

`public virtual String ToString()` returns a string representation of the object

- Tip: all descendants should override this method
- Tip: the result should be a concise and informative representation

Arrays

- Arrays are objects of class **`System.Array`**
  - but with the familiar syntax to access them
- Operator [ ] to access components
- Field **`Length`** denotes the number of elements
- All components must a have a “common” type
  - a common ancestor in the inheritance hierarchy
- Array components are automatically initialized to defaults
- Three variants
  - Monodimensional: **`string[] arr;`**
  - Multidimensional: **`string[,,] arr_3d;`**
  - Jagged (array-of-arrays): **`string[][]  aOfa;`**
    - **`aOfa[i]`** is a reference to a mono-dimensional array

Array use

```
// mono-dimensional of size 7
int[] iArray = new int[7];
// multi-dimensional of size 2x5x8
int[,,] mdArray = new int[2,5,8];
// jagged
int[][] jArray = new int[2][];
// using initializers
Vehicle[] v1 = {new Car(), new Truck()};
int[,] mdArray = {{1,2}, {3,4}, {5,6}};
int[][] jArray = new int[]{new int[] {0, 1},
                           new int[] {8,7,6}};
```

Enumerated types

Denote a finite set of named integer values

```
enum TypeName : intType {VALUE_1, ..., VALUE_N};
```

- `intType` defaults to `int` if omitted
- Enumeration starts from 0 by default, with step 1
- Can define different values: `VALUE_3 = 8;`

Within the type system, `TypeName` is a class that extends class `System.Enum` and with `N` static integer values

```
TypeName aValue = TypeName.VALUE_2;
```

Unlike in Java, C#'s `enum` does not define a full-fledged class with constructors, fields, etc.

Enumerated types (cont'd)

Convenient way to define a set of integer constants

```
enum Days {Monday, Tuesday, ...};
```

An `enum` declaration defines a type with limited capabilities

- Variable instantiation:

```
Days d = new Days();  // d has value 0
```

- Can refer to elements of enumeration:

```
d = Days.Monday + 3;  // d has value 0+3
```

Default initialization of `enum`'s without 0 yields undefined behavior:

```
enum Parity {odd = 1, even = 2};
Parity p = new Parity();
```

The default initialization of `p` is to the default `int` value 0:

```
p = (Parity) 0;  // allowed but undefined!
```

Structs

Structs are a sort of “lightweight classes”

- mostly supported for continuity from C/C++

Can have fields, methods, and other features of classes

Important differences between structs and full-fledged classes

- **`struct`**s define value types: they are stack-allocated
  - difference if passed as method arguments
- if constructors are present, they must have arguments
- can be instantiated without **`new`** (by directly setting fields)
  - but then cannot be used until all fields are initialized
- can implement interfaces
- cannot inherit from another **`struct`** or class

- Tip: if you need methods and constructors, you’d probably better use a class

Properties

Properties are shorthands to define pairs of setter and getter for a field

- Properties are syntactic sugar to facilitate proper encapsulation

A property has a name and a type

- For a client of the class, a property is indistinguishable from a field with the same name

A property can have a setter, a getter, or both

- Keywords: **`set`**, **`get`**
- Within a setter: **`value`** refers to the value passed to the setter
- A property can also be **`static`**

Properties: example

```
public class Employee {
  private int empAge;
  public int Age {
     get { return empAge; }
     set { empAge = value; }
  }
}
```

Usage:

```
Employee e = new Employee();
e.Age = 33;     // calls setter with value==33
int a = e.Age; // calls getter
```

Properties: example (2)

```
public class Employee {
  private int empAge;
  public int Age {
      get { return empAge; }
      set { empAge = value; }
  }
}
```

This straightforward implementation of properties is equivalent to the default:

```
public class Employee {
  public int Age { get; set; }
}
```

Indexers

Indexers are similar to properties, but for “indexed” fields

- typically arrays (and possibly other maps)

An indexer has a type and an index argument

- no specific name

```
public class ATPRanking {
  private string[] list = new string[1000];
  public string this[int pos] {
   get { if (1 <= pos && pos <= 1000)
         return list[pos-1]; else return “”; }
    set {if (1 <= pos && pos <= 1000)list[pos-1]=value;}
   }
}
```

Usage:

```
ATPRanking r = new ATPRanking();
r[8] = “Roger Federer”;      // calls setter
string n = r[100];           // calls getter
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Assertions and contracts

Contracts

Contracts are specification elements embedded in the program text. They use the same syntax as Boolean expressions of the language. Here’s an example with Eiffel syntax.

```
class BankAccount

  balance: INTEGER

  deposit (amount: INTEGER)
    require amount > 0                      // precondition
    do balance := balance + amount
    ensure balance > old balance end // postcondition

invariant
  balance >= 0                              // class invariant
end
```

Contracts: preconditions

The precondition of a method **M** specifies requirements that every call to **M** must satisfy. It is the caller's responsibility to ensure that the precondition is satisfied.

```
ba: BankAccount
create ba                       // object creation

ba.deposit (120)                // valid call: 120 > 0
ba.deposit (-8)                 // invalid call: -8 < 0
```

Contracts: postconditions

The postcondition of a method **M** specifies conditions that hold whenever an invocation to **M** terminates. **M**'s body is responsible to ensure that the postcondition is satisfied.

```
ba: BankAccount
create ba                    // object creation
// assume 'balance' is 20

ba.deposit (10)
      // postcondition ok: 30 > 20
ba.deposit (MAX_INTEGER)
      // postcondition violation if balance
         silently overflows into the negatives
```

Contracts: class invariants

The class invariant of a class **C** constrains the states that instances of the class can take. The class invariant's semantics is a combination of the semantics of pre- and postcondition: the class invariant must hold upon object creation, right before every qualified call to public members of **C**, and right after every call terminates.

```
ba: BankAccount
create ba                    // object creation
// class invariant must hold

// class invariant must hold
ba.deposit (10)
// class invariant must hold
```

Assertions

The .NET framework offers assertions: checks that can be placed anywhere in the executable code:

```
Debug.Assert(boolean-expr)
```

Assertion checking is enabled only in debug builds:

- If evaluates to true, nothing happens
- If evaluates to false, the run is interrupted and control returns to the debugger

We can use assertions to render the semantics of contracts.

Code Contracts

Since version 4.0, the .NET framework includes full support of contracts through CodeContracts

- Preconditions, postconditions, class invariants
- Runtime checking (exception mechanism)
- Static checking
- Documentation generation

CodeContracts are offered as a library rather than natively

- Advantage: available across the .NET platform
- Disadvantage: verbose syntax

Currently only partially supported in the Mono platform

More information:

http://research.microsoft.com/en-us/projects/contracts/

Code Contracts: example

```
using System.Diagnostics.Contracts;

class BankAccount {

  int balance;

  void deposit (int amount) {
    Contract.Requires (amount > 0); // precondition
    Contract.Ensures                // postcondition
      (balance > Contract.OldValue <int>(balance));
    balance += amount;
  }

  [ContractInvariantMethod] // class invariant
  void ClassInvariant()
    { Contract.Invariant(balance >= 0); }
}
```